Adlib: A Distributed Array Library to Support HPF Translation

D.B. Carpenter

High Performance Computing Centre,
University of Southampton,
Southampton, S09 5NH

DRAFT

Abstract

Adlib is a class library implementing the HPF distributed data model. Its native interface is in C++. A Fortran interface provides a subset of the Adlib functionality. This paper mainly discusses the C++ interface and implementation.

1 Introduction

The *Adlib* communication library is a component of the Southampton HPF translation system. It combines ideas from the ADAPT system for translation of Distributed Fortran 90, and from earlier collective communication libraries developed at Southampton. Its features include

- Extensive use of abstract data types used to describe process arrays, distributed data, distributed control, and collective communication in an integrated way.
- A communication functionality adequate to fully support translation of distributed Fortran.
- Data distribution formats to support full HPF.

Adlib is implemented as a C++ class library. There is a reduced interface to Fortran.

2 Process Arrays

The HPF declaration

```
!HPF$ PROCESSORS P (4, 4)
```

introduces a 4 by 4 arrangment of virtual processors, over which program data can be distributed. In Adlib the role of the HPF processor arrangement is taken by the *process array*. A rank 2 process array, equivalent to `P`, is declared by

```
Procs2 p(4, 4) ;
```

An Adlib process array has a set of *process dimensions* of class `Dimension`. The individual dimensions of a process array are accessed by the member function `dim`, illustrated in the next section.

A `Procs` array describes *virtual processes*, which are mapped (somehow) into the available set of physical processes. In the remainder of this paper we will occasionally be cavalier about the distinction between virtual and physical processes; if the library in used as intended the distinction should not be significant.

A process array is a particular case of a *process group*. Adlib has a more general idea of a process group, to be described in section ??.

3 Ranges

A central idea in Adlib is the *range*. There is no completely analogous object in HPF. A range is a one-dimensional index space. A range of extent n has the values $0 \ldots n-1$ associated with it. The range defines a mapping from this interval into a process dimension.

A *primitive range* is distributed over a process dimension, according to one of the HPF distribution formats. The statements

```
Range x(100, p.dim(1), BLK) ;
Range y(200, p.dim(2), CYC, 10) ;
Range z(50) ;
```

declare that `x` has extent 100, and is distributed block-wise over the first dimension of `p`; `y` has extent 200 and is distributed block-cyclically over the second dimension of `p`. `z` is a "collapsed" range of extent 50. An Adlib primitive range is directly analogous to a single dimension of an HPF template.

A *derived range* is a range linearly aligned to some primitive range. The statements

```
Range i(20, 0, 5, x) ;
Range j(100, 10, 1, y) ;
Range k(50, 74, -1, j) ;
```

declare that `i` has extent 20; it is aligned to `x` starting at position 0 in `x`, with stride 5—that is, element 0 of `i` has the some process mapping as element 0 of `x`, element 1 of `i` has the same mapping as element 5 of `x`, etc. `j` has extent 100; it is aligned to `y` starting at postion 10 with stride 1. `k` has extent 50; it is aligned to `y` starting at position 84 with stride -1 (the immediate parent of `k` is `j` but the two alignments are composed to yield the mapping to the primitive range—two derived ranges are equivalent if they have to same extent and the same alignment to equivalent primitive ranges, however they are derived).

The extent of a range can be determined by the member function `size`. Its interface is

```
int Range :: size() ;
```

4 Arrays

An *array* is distributed over a particular process group. The shape and mapping of an array of rank r are defined by list of r ranges. Each of these ranges must be distributed over a dimension of the array's process group, or it must be collapsed; no two of them may be distributed over the same dimension.

```
Array2<float> a(p, x, y) ;
Array1<float> c(z) ;
```

The (optional) first argument of the constructor is the process group over which the array is distributed. If it is omitted, as in the declaration of `c`, the array is distributed over the active process group (see section ??).

According to these declarations `a` is a rank 2 array of floats, shape `(100, 200)`, distributed over the process array `p`. `c` has a collapsed range and is replicated over the active process group. If no range of an array is distributed over an effective dimension of its process group, the array is replicated over that dimension. `c` has no distributed ranges so it must be replicated over all dimensions of the active process group.

Creation of an array is a collective operation—all members of the active process group must cooperate in the construction.

5 Local subscripts

Adlib arrays can be subscripted by objects from the `Subscript` class. This is as an abstraction of the local subscripts for distributed array access. A typical implementation would incorporate a representation of a coordinate within a process array, and a local array subscript.

A `Range` object functions as a map between an integer global array subscript and the `Subscript` class. So, for example

```
Subscript s(x(30)) ;
```

declares `s` as a local subscript assocated with element 30 of the distributed range `x`. The expression is ill-defined if and the process which evaluates it does not hold a copy of this element of the distributed array.

Elements of array may be accessed by passing appropriate `Subscript` arguments to the (overloaded) function application operator on the array. So

```
a(x(30), y(50))
```

returns a reference to element `(30, 50)` of `a`.

It is not required that a range used to construct a subscript is identical to the corresponding range of the array, but it should be *derived* from that range. Because of the alignment between `i`, `j` and `x`, `y` the subscript `u` defined by

```
Subscript u(i(6)) ;
```

is equivalent to `s` and

```
a(i(6), j(40))
```

selects the same array element as the earlier expression.

6 Embedding control: *on* and *at*

As explained in the last section an array element may only be referenced by a process which holds a copy of the element. The library provides two constructs which can be used to test whether a subscript is held locally.

The function `on` is a member of the process group classes (including the process array classes). It determines whether the local physical process holds any elements of the virtual process group

```
if(p.on()) {
  ...
} p.no() ;
```

The code represented by the ellipsis is only executed by physical processes which hold part of the process array `p`. Logically, `no` returns control to the whole of the original active process group. This logic will explained further in section ??.

Having established that some members of the virtual process array is held locally, one proceeds to test whether subscripts embedded in individual process dimensions are held locally using the `at` member of the process group class. If `s` is a local subscript, the idiom is

```
if(s.at()) {
  ...
} s.ta() ;
```

Again, we defer explanation of the need for the matching `ta` call to section ??.

Collecting together various fragments introduced so far, we can write a complete program which modifies an element of a distributed array.

```
Procs2 p(4, 4) ;

Range x(100, p.dim(1), BLK) ;
Range y(200, p.dim(2), CYC, 10) ;

Array2<float> a(p, x, y) ;

Subscript s(x(30), t(y(50)) ;

if(p.on()) {
  if(s.at()) {
    if(t.at()) {
      a(s, t) = 73 ;
    } t.ta() ;
  } s.ta() ;
} p.no() ;
```

The program assigns the value 73 to element `(30, 50)` of `a`.

The user has to exercise some care when updating array elements. It is important to ensure that elements which are meant replicated over a process group (according the mapping implied by the array declaration) are updated with a consistent values. It is easy enough to devise a set of variable usage conventions which will ensure this requirement is met—these conventions will be omitted here.

7 Distributing control: *where*

If `at` is like a conditional, `where` is like a loop. Instead of a subscript it is parametrised by a range. Consider the following fragment

```
if(p.on()) {
  for(x.where() ; x.test() ; x.next()) {
    for(y.where() ; y.test() ; y.next()) {
      a(x, y) = x + y ;
    } y.erewh() ;
  } x.erewh() ;
} p.no() ;
```

The `where` member activates a `Range` as an "iterator". The loop body executes once for every index value of `x` held on the local process.

The assignment in the body illustrates two type-conversions defined on the `Range` class. A `Range` can be cooerced to an integer, which is how `x` and `y` are used on the right hand side of the assignment. It can also be cooerced to a `Subscript`, which is how they are used on the left hand side.

If these cooercions appear while a range is activated as an iterator, cooercion to an integer returns the range index associated with the current loop iteration, while cooercion to a local subscript returns a subscript embedded at the corresponding location.

The effect of the parallel loop above is to assign to each element of `a` the sum of its two index values.

8 Communication

There are three families of communication operations in Adlib

8.1 The *remap* family

The `remap` operation copies an array to another array of the same shape. Its interface is

```
void remap(Array dst, Array src) ;
```

The mappings of the source and destination arrays are arbitrary. In common with all the communication routines described in this paper, `remap` is a collective operation—it must be invoked on all processes (in the active process group cf section ??).

We will now give a small example of an Adlib program. The HPF program in figure 1 is a toy example of the nearest neighbour update paradigm. The Adlib program in 2 is an equivalent Adlib program. Construction of the array sections and declaration of temporaries has been lifted out of the loop over `n`

The *remap* family potentially contains other operations which can be expressed in terms of `remap`, but may be more efficiently implemented directly. These include broadcasts, shifts and matrix transpose.

8.2 The *gather-scatter* family

The general `gather` operations have interfaces

```
void gather(Array b, Array a, Array i_1) ;

void gather(Array b, Array a, Array i_1, Array i_2) ;

void gather(Array b, Array a, Array i_1, Array i_2, Array i_3) ;
...
```

```
real u(10)

!hpf$ distribute u (block) onto p
!hpf$ processors p (4)

integer i, n

do i = 1, 10
  if(i .eq. 1 .or. i .eq. 10) then
    u(i) = 1.0
  else
    u(i) = 0.0
  endif
enddo

do n = 1, 5
  u(2 : 9) = 0.5 * (u(1 : 8) + u(3 : 10))
enddo
```

Figure 1: Simple nearest neighbour update in HPF.

Here `b` is the *gathered data* and `a` is the *scattered data*. These are followed by subscript arrays, which are integer arrays with the same shape and mapping as the gathered data. The number of subscripts should be equal to the rank of the scattered data, and their values should lie in the ranges expected by subscripts of that array. Data is copied from `a` to `b` with the specified subscripting pattern. A completely analogous primitive `scatter` copies data from `b` to `a`.

These operations are unnecesarily general if it is only required to implement the patterns of access in Fortran 90 array syntax (but they are needed for parallelising `FORALL`, for example). Vector subscripts can be handled by the simpler primitives

```
void vecGather(Array b, Array a, Array i_1) ;

void vecGather(Array b, Array a, Array i_1, Array i_2) ;

void vecGather(Array b, Array a, Array i_1, Array i_2, Array i_3) ;
...
```

where now `a` and `b` have the same rank, and the subscripts are rank-1 arrays, whose individual ranges are the same as the corresponding to range of `b`. If any subscript argument is `NULL` it is assumed that the gathered and scattered data

```
Procs1 p(4) ;

if(p.on()) {
  Range x(10, p.dim(1), BLK) ;

  Array1<float> u(x) ;

  for(x.where() ; x.test() ; x.next()) {
    if(x == 0 || x == 9)
      u(x) = 1.0 ;
    else
      u(x) = 0.0 ;
  } x.erewh() ;

  Range l(8, 0, 1, x), r(8, 2, 1, x) ;

  Array1<float> u_l(u, l), u_r(u, r) ;

  Range c(8, 1, 1, x) ;

  Array1<float> tmp1(c), tmp2(c) ;

  for(int n = 1 ; n <= 5 ; n++) {
    remap(tmp1, u_l) ;
    remap(tmp2, u_r) ;

    for(c.where() ; c.test() ; c.next()) {
      u(c) = 0.5 * (tmp1(c) + tmp2(c)) ;
    } c.erewh() ;
  }
} p.no() ;
```

Figure 2: Simple nearest neighbour update in Adlib.

have the same extent in the associated dimension, and the operation behaves like an array assignment with respect to that dimension. `vecScatter` is similar.

8.3 The reduction family

The simplest reduction primitive has the inteface

```
void reduce(const int typeLen, char* res, Frame* src,
            void (*opZero)(void*),
            void (*opAccum)(void*, void*)) ;
```

The cast to a character vector of length `typeLen`, is replicated over the active process group. The array `src` holds the input data; the accumulation operation is defined by the function `opAccum`, and the function `opZero` sets data of the return type to the zero of the accumulation operation.

Variants do reductions on pairs of arrays (scalar product operations and masked reductions) and reductions in single array dimensions, where the result is an array with rank reduced by one.

9 Process groups

Having provided a few examples of working code, we turn to an idea which is important to deeper understanding of Adlib—the process group.

A *process group* is a set of (virtual) processes. The process group has two röles—a particular action can be performed within a specified process group, or a particular data object can be distributed over a specified process group.

The simplest example of a process group is a process array. Adlib has a more general idea of a process group, which is a "slice" of a process array formed by restricting one or more of the process dimensions to a single coordinate.

A process coordinate can be defined implicitly by a subscript. The operator / is overloaded to construct a new process group (class `ProcessGroup`) from an existing one and a subscript.

```
ProcessGroup q(p / s) ;
ProcessGroup r(q / t) ;
ProcessGroup s(p / s / t) ;
```

The process groups `q`, `r` and `s` are illustrated in figure ??. The *parent process array* of all these groups is `p`. A process group has a set of process dimensions taken from the parent process array. The dimension set for `p` contains `p.dim(1)` and `p.dim(2)`. The dimension set for `q` contains just `p.dim(2)`, while that for `r` contains just `p.dim(1)`. The set for `s` is empty.

An immediate benefit of this generalised process group is that it allows us to extend mapping of arrays in a way necessary to fully support HPF. HPF allows "embedded" alignments in templates.

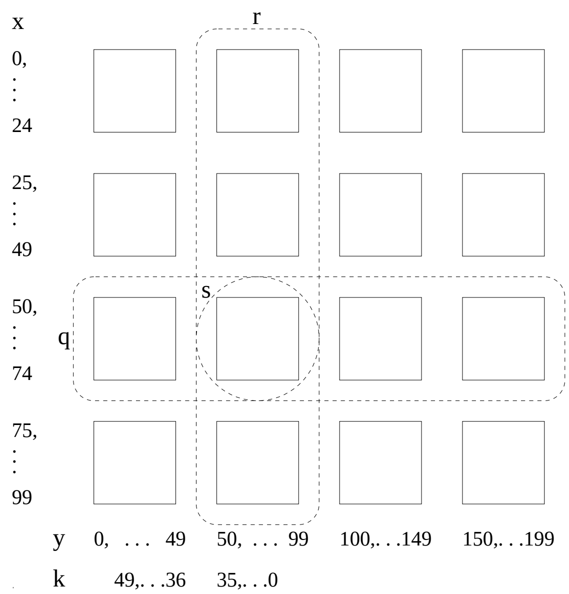

Figure 3: Examples of process groups. The square boxes represent the 16 virtual processes in the process array p. The dashed lines embrace groups q, r and s.

```
!$HPF PROCESSORS p(2, 2)

!$HPF TEMPLATE t(10, 10)
!$HPF DISTRIBUTE t(BLOCK, BLOCK) ONTO p

INTEGER b(10)
!$HPF ALIGN b(:) WITH t(:, 5)
```

The Adlib equivalent is

```
Procs2 p(2, 2) ;

Range x(10, p.dim(0), BLK) ;
Range y(10, p.dim(0), BLK) ;

Array1<int> b(p / y(4), x) ;
```

In section 4 the first argument of the array constructor was supposed to be the process *array* over which the data is distributed; in general it can be an process *group*. The ranges of the array must be distributed over dimensions of this group.

10 The active process group

The second benefit of the generalisation is that it supports representation of the "active process group" inside arbitrary `at` and `where` constructs. We proceed to define this idea more carefully.

At any stage of execution of a program a particular process group is singled out as the *active process group* (APG). From the viewpoint of a particular process, the active process group is the set of processes sharing the current "thread of control". Adlib supports MIMD execution through the `where` construct, so there may be several distinct processes groups active at the same time, with the physical processes partitioned between these groups.

A powerful feature of Adlib, designed to support "nested parallelism", is that collective operations such as communication operations, array declarations, and even process array declarations, can appear at any point of the program. It is *not* required that all physical processors engage in the operations; it is sufficient that all processes in the active process group do so. Any global synchronisation implied by the collective operation affects just the processes in the active group, leaving processes outside the group undisturbed [1].

The three Adlib control constructs affect the active process group in the following ways. If `p` is a process group, inside

```
if(p.on()) {
  ...
} p.no() ;
```

the active process group is `p`. If the active process group is `p` and `s` is a subscript, inside

```
if(s.at()) {
  ...
} s.ta() ;
```

the active process group is `p / s`. If the active process group is `p` and `x` is a range, inside

```
for(x.where() ; x.test() ; x.next()) {
  ...
} x.erewh() ;
```

the active process group is `p / (Subscript) x`—employing the cooercion from range to subscript described in section 7.

From these formulae we infer two important restrictions which can be expressed in terms of the active process group

[1] This requirement molds the idiom for the three control constructs. The operations `on`, `no`, `at`, `ta`, `where` and `erewh` update a hidden data structure which describes the active process group. The data structure is interrogated by various collective communication operations. Although `no`, for example, does not return a value, it does have the important side effect of restoring this data structure to the state it had before the matching `on` call.

- An *at* construct can appear if and only if its subscript is derived from a range distributed over a dimension of the active process group.
- A *where* construct can appear if and only if its range is distributed over a dimension of the active process group.

11 Accessibility

There are some some restrictions on the arguments of collective operations. These restrictions can be expressed in terms of "accessibility". A process group q with parent array p is *accessible* at a particular stage of program execution if the active process group is the same as the one in effect when p was created, or if it is a subarray of p which contains q. For example

```
Procs2 p(2, 2) ;

Range x(10, p.dim(0), BLK) ;
Range y(10, p.dim(1), BLK) ;

Subscript s(x(5)) ;
Subscript t(y(5)) ;

ProcessGroup q(p / s) ;

// Point A

if(p.on()) {
  // Point B

  if(s.at()) {
    // Point C

    if(t.at()) {
      // Point D
    } t.ta() ;
  } s.ta() ;
} p.no() ;
```

The group q is accessible at point A because the APG is the same as the one in effect when p was defined. It is accessible at point B because the APG is a subarray of p (viz p itself) which contains q. It is accessible at point C because the APG is a subarray of p (this time q itself) containing q. The group q is *inaccessible* at point D.

Informally the idea is that a process group is accessible if that group is contained within the APG, with the intuitive understanding that a process array is contained within the group active at the time it was declared.

Two important restrictions can be expressed in terms of accesibility:

- An *on* construct can appear if and only if its process group is accessible.
- In general a collective operation on distributed data can appear only if all its arguments are distributed over accessible process groups.

12 Arrays and the active process group

In section 4 we gave an illustration of an array in which the process group was omitted

```
Array1<float> c(z) ;
```

and stated that `c` was globally replicated with a collapsed range. In general if the process group argument is omitted the array is distributed over the active process group. It need not be replicated. It may have ranges distributed over this group, as in

```
Procs2 p(2, 2) ;

if(p.on()) {
  Range x(10, p.dim(0), BLK) ;
  Range y(10, p.dim(1), BLK) ;

  Array2<float> u(x, y) ;
  Array1<int> v(x) ;

  // ...
} p.no() ;
```

The arrays `u` and `v` are distributed over `p`. The latter array is replicated in the second dimension.

13 Array sections and process groups

In general the subscript arguments of the array section constructor are a mixture ranges and local subscripts. The section is distributed over the process group of the parent array, restricted by any local subscript arguments. So in

```
Procs2 p(2, 2) ;
```

```
Range x(10, p.dim(0), BLK) ;
Range y(10, p.dim(1), BLK) ;

Array2<int> a(p, x, y) ;

Array1<int> b(a, x, y(5)) ;
```

the section `b` is distributed over `p / y(5)`.

If a range currently activated as a **`where`** iterator is used as a section subscript argument, it will (in effect) be cooerced to a local subscript.

Uniquely amongst collective operations, the parent array argument of the array section constructor need not be accessible at the point of call. It is only required is that the *constructed* array be distributed over an accessible process group. With the declarations above we could have

```
Subscript s(y(5)) ;

if(s.at()) {
  Array1<int> b(a, x, s) ;

  // ...
} s.ta() ;
```

The process group `p`, over which `a` is distributed, is not accessible at the point where `b` is constructed. The declaration is legal because the process group `p / s` over which `b` is distributed *is* accessible (it is the current APG).